ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

COMUNICACIÓN MULTIMAESTRO A TRAVÉS DE PAR TRENZADO RS-485 (HALF-DUPLEX)

MULTIMASTER COMMUNICATION THROUGH RS-485 TWISTED PAIR (HALF-DUPLEX)

Realizado por
Ángel de Jesús García Pineda
Tutorizado por
Juan Carlos Tejero Calado
Departamento
Electrónica

UNIVERSIDAD DE MÁLAGA
MÁLAGA, DICIEMBRE 2015

Fecha defensa:
El Secretario del Tribunal

Resumen:

En este proyecto se ha desarrollado un protocolo para establecer una comunicación multimaestro entre distintos dispositivos sobre una red de cable de par trenzado rs-485 de tipo half-duplex.
Se explican los modos de envío half-duplex y full-duplex y los modelos de comunicación multimaestro y maestro-esclavo para establecer diferencias y se justifica la implementación de este protocolo.
Se ha realizado un estudio y análisis del protocolo a implementar y se describen las decisiones de diseño empleadas para resolver los distintos problemas encontrados a la hora de analizar la capacidad de la red.
Luego se ofrece una implementación del protocolo en lenguaje C/C++ para la plataforma Arduino y se explican los distintos casos de uso que se pueden dar por dispositivo.
En este documento se describe de forma completa y punto por punto el trabajo realizado a lo largo de varios capítulos en forma de texto e imágenes o representaciones que dan al lector distintas vías para entender lo que aquí se explica.

Palabras claves:

rs485, rs-485, half-duplex, par trenzado, multi-maestro, multimaestro, multi maestro, protocolo, domótica, red, mensajes, transmisión, dispositivos, arduino

Abstract:

In this project, a protocol has been developed to establish a multimaestro communication between different devices on a twisted pair rs-485 half-duplex type cable network.
Explains ways of shipping half-duplex and full-duplex and multimaestro communication and master-slave models to establish differences and justified the implementation of this protocol.
He has carried out a study and analysis of the Protocol to implement and describe the design decisions used to solve various problems encountered when analyzing the capacity of the network.
Then provides a protocol implementation in c/c++ language to the Arduino platform and explains the different use cases that may occur by device.
This document describes fully and point by point the work carried out over several chapters in the form of text and images or representations which give the reader ways to understand what is explained here.

Keywords:

rs485, rs-485, half-duplex, twisted pair, multi-master, multimaster, multi master, protocol, home automation, network, messages, transmission, devices, arduino

Índice general

Capítulo 1

Introducción

1.1 Estándar RS485

RS485 (también conocido como EIA485) es un estándar de comunicaciones en bus de la capa física del Modelo OSI desde 1983.

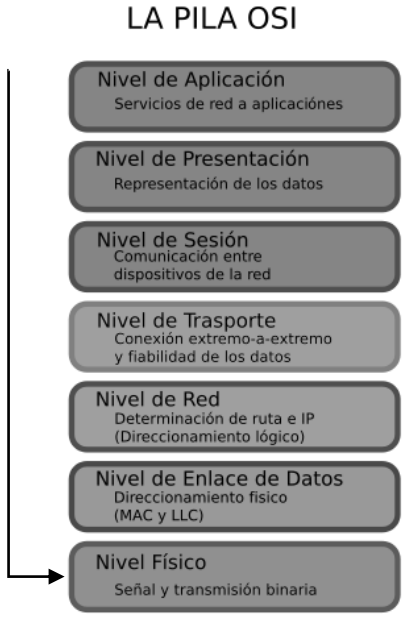


Figura 1. Modelo OSI [Ref 9]

Está definido como un sistema de bus diferencial multipunto apto para transmitir datos a grandes distancias y para operar en ámbitos eléctricamente ruidosos.

Permite una velocidad de datos de 35 Mbit/s a 10 metros de distancia y de 100 Kbit/s cuando se conectan dispositivos separados hasta 1200 metros entre sí.

Aunque inicialmente permitía hasta 32 dispositivos en la red, existen interfaces que permiten hasta 256 dispositivos.

Esto nos va a permitir enviar desde pequeñas a grandes cantidades de datos entre dispositivos que puedan estar relativamente cerca o realmente lejos a una alta velocidad. Esto lo hace ideal para todo tipo de instalaciones donde los dispositivos tengan que estar interconectados, como por ejemplo la automatización de una casa o la gestión de una red de sensores en cualquier ámbito.

1.1.1 Cable de par trenzado

Se hace uso de cable de par trenzado, lo que lo hace también ideal para instalaciones donde pueda generarse ruido que afecte negativamente a la transmisión, como en una instalación industrial.

El cable de par trenzado reduce dos de los mayores problemas para los diseñadores de comunicaciones de alta velocidad y largas distancias: las interferencias electromagnéticas recibidas y las interferencias electromagnéticas generadas.

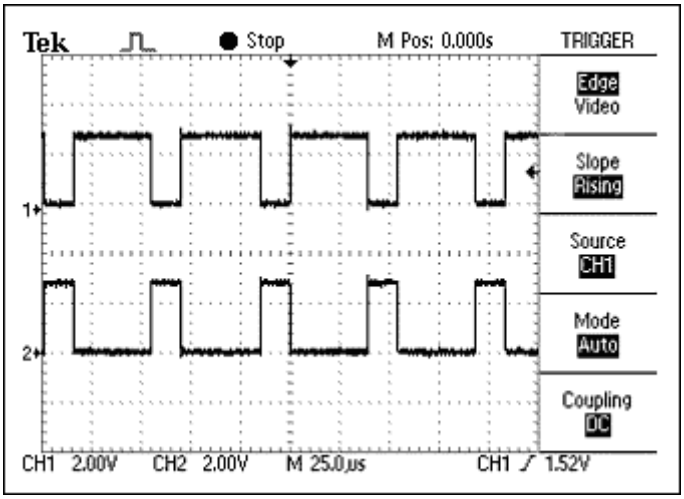


Figura 2. Señales idealmente opuestas en ambos hilos [ref 8]

Un cable de par trenzado está formado por dos hilos y por ambos viaja la misma información, aunque desfasada 180º en un cable respecto al otro. De esta manera conseguimos que cualquier interferencia que pudiera introducirse en el cableado lo hará en ambos hilos por igual, con la misma polaridad y amplitud.
Posteriormente, en el destino de los datos, las señales se restituirán en polaridad y los picos de ruidos que se habían llegado a introducir, al invertirse las señales, se neutralizan y se eliminan entre sí, recuperando así la señal inicial transmitida.

1.1.2 Modos de envío: Half-Duplex y Full-Duplex

La comunicación en RS485 fue diseñada para que sólo pueda haber un dispositivo transmitiendo por cada cable de par trenzado. Permite modos de envío *half-duplex* y *full-duplex*. En la figura 3 se puede apreciar la diferencia entre ambos modos.

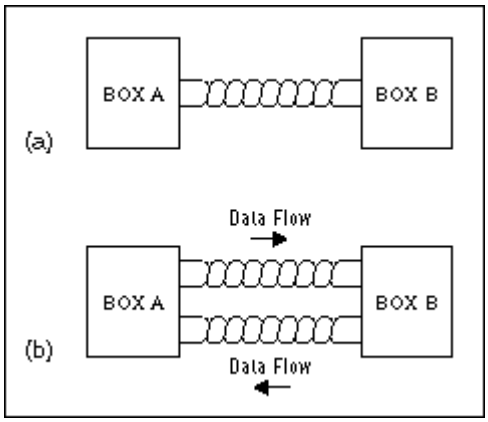


Figura 3. Half-duplex (a) y full-duplex (b) [Ref 7]

Del modo *half-duplex*: el dispositivo A puede transmitir datos al dispositivo B y el dispositivo B puede transmitir datos al dispositivo A pero no pueden transmitir ambos al mismo tiempo. Sólo es necesario un cable de par trenzado.

Del modo *full-duplex*: tanto el dispositivo A como el dispositivo B pueden transmitir entre ellos simultáneamente. En este caso es necesario el uso de dos cables de par trenzado.

1.2 Modelos de comunicación

Hay multitud de modelos de comunicación pero los más usados y los que vamos a tratar son el Maestro-Esclavo, comúnmente usado en redes tipo *half-duplex*, y el Multimaestro, que es comúnmente usado en redes *full-duplex*.

1.2.1 Maestro-esclavo

Maestro-esclavo (Master-slave en inglés) es un modelo de protocolo de comunicación en el que un dispositivo o proceso (conocido como maestro o *master*) controla uno o más de otros dispositivos o procesos (conocidos como esclavos o *slaves*).

En este tipo de comunicación, el dispositivo maestro es siempre quién tiene toda iniciativa de comunicación en detrimento de los dispositivos considerados esclavos que lo único que pueden hacer es esperar una petición del maestro para poder responder a éste.

Para ser más precisos, el dispositivo maestro administra la red y actúa de mediador, de forma que pregunta periódicamente a cada dispositivo esclavo para ver si necesita realizar alguna acción determinada. El dispositivo esclavo podrá responder con una petición al maestro si necesita algo. Esto se puede apreciar en el diagrama de secuencia siguiente, figura 4.

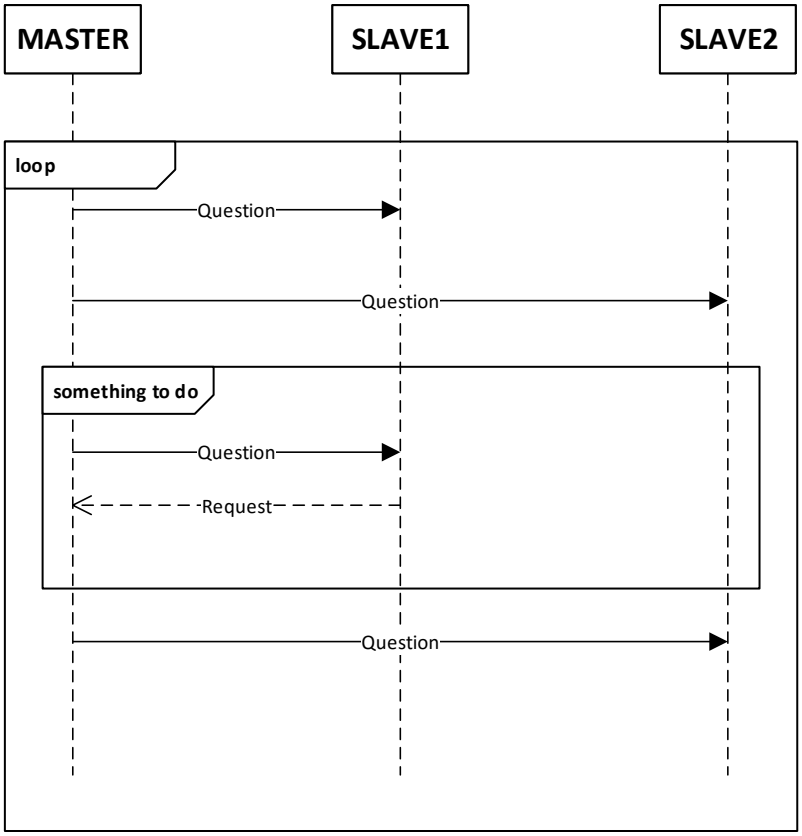


Figura 4. Diagrama de secuencia maestro-esclavo

Ningún modelo es perfecto y éste tampoco lo es, una desventaja es que es imprescindible tener un dispositivo en la red que actúe como maestro y si por alguna razón su funcionamiento fuera erróneo, toda la red quedaría paralizada.

Además es lento pues tiene que ir preguntando a todos los dispositivos si tienen algo que hacer y luego esperar una respuesta de cada uno de ellos durante un tiempo predeterminado.

Y aún en el caso de que uno de los dispositivos esclavos quisiera realizar alguna acción, puede que necesitara algo de algún otro dispositivo, lo que aumenta el tiempo de respuesta, como en este ejemplo:

1º - El dispositivo maestro pregunta al dispositivo esclavo (1) si necesita algo.

2º - El dispositivo esclavo pide un dato que sólo conoce otro dispositivo esclavo (2).

3º - El dispositivo maestro pide al otro dispositivo esclavo (2) el dato.

4º - El otro dispositivo esclavo (2) responde al dispositivo maestro con el dato.

5º - El dispositivo maestro responde al dispositivo esclavo (1) con el dato requerido.

Como ventaja podríamos decir que es el modelo ideal para las redes de tipo *half-duplex*, porque se asegura que sólo habrá un dispositivo transmitiendo al mismo tiempo.

1.2.2 Multimaestro

El modelo multimaestro se basa en el modelo maestro-esclavo ya que, sólo puede haber un dispositivo con el rol de maestro, es decir, con capacidad de transmitir en un momento determinado. La diferencia reside en que cualquier dispositivo de la red puede ser, además de esclavo, maestro a la hora de querer pedir o responder.
Inicialmente todos los dispositivos en la red son esclavos y se le llama maestro al dispositivo que toma el bus en ese instante.

Con este modelo evitamos que exista una figura imprescindible y necesaria dentro de la red, cualquiera puede tomar el control del bus y transmitir a otro dispositivo de la red, sin intermediarios. En la figura 5 se puede apreciar a grandes rasgos un funcionamiento ideal.

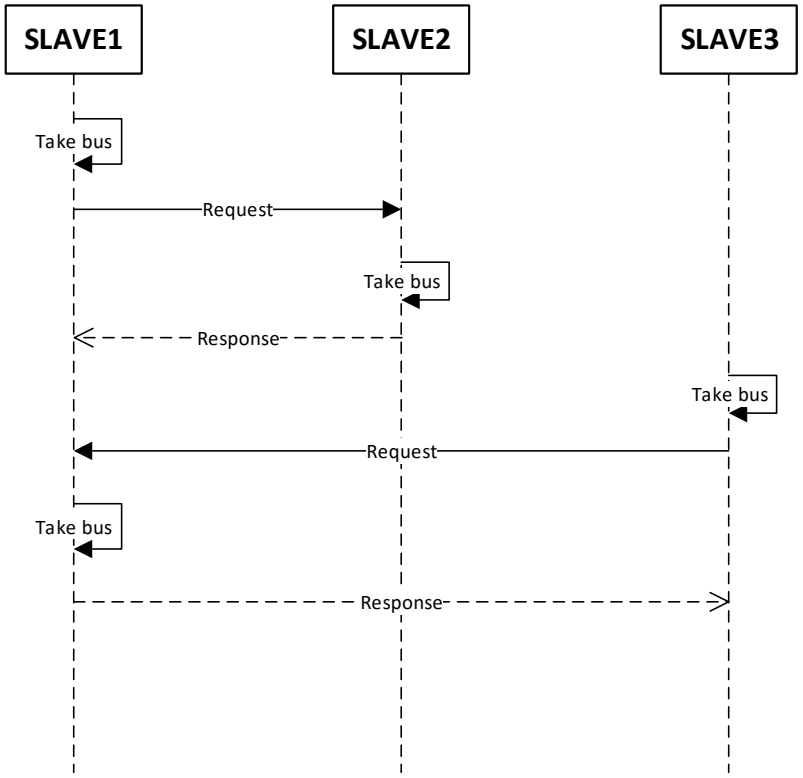


Figura 5. Diagrama de secuencia multimaestro

Obviamente, como todos, no es perfecto y también tiene desventajas, específicamente en el tipo de red *half-duplex* donde sólo un dispositivo puede transmitir a la vez y donde, por tanto, habría que gestionar y arbitrar el acceso al medio para tratar de evitar envíos simultáneos o colisiones.

1.3 Motivación y objetivos

La principal motivación de este proyecto es la de conseguir darle otro uso al tipo de cable utilizado en redes RS485 que pueden ser utilizadas en diferentes tipos de instalaciones y que por precio y características no tiene nada que envidiar a otro tipo de cableado o sistemas de transmisión.

El objetivo de este proyecto ha sido la implementación de un protocolo para establecer una comunicación, de tipo multimaestro por cable de par trenzado bajo el estándar RS485, con la que conectar múltiples dispositivos a través del mismo bus y hacer de ellos dispositivos totalmente independientes en la red, salvando las dificultades derivadas como son las colisiones o envíos simultáneos.

A lo largo de este proyecto se ha procedido al estudio de los estándares y protocolos ya existentes, así como de bibliotecas que pudieran ser utilizadas para el desarrollo de una solución.

Finalmente no todo se ha limitado a la implementación genérica del protocolo o biblioteca, se ha desarrollado una prueba de concepto del protocolo para Arduino, aunque el protocolo es implementable en cualquier otra plataforma.
Posteriormente se probado con un sistema completo con sensores y actuadores sobre Arduino con el que depurar y hacer las pertinentes pruebas.

1.4 Sobre la memoria

Se ha estructurado la memoria de la siguiente manera:

- En el primer capítulo se hace un resumen del estándar RS485, de los modos de envío *half-duplex* y *full-duplex* y de los modelos de comunicación. También cual ha sido la motivación para desarrollar este trabajo, los objetivos propuestos y la organización de esta memoria.

- En el segundo capítulo se realiza un estudio y análisis del protocolo que implementaremos y se describen las decisiones de diseño empleadas.

- En el tercer capítulo se explica cómo se ha realizado la implementación final de la librería con lo explicado en el segundo capítulo, los distintos casos de prueba que se pueden dar y como hacer uso de ella sobre un sistema, en este caso Arduino.

- En el cuarto y último capítulo se comentan las conclusiones a las que se ha llegado tras la realización de este proyecto.

A lo largo de la memoria se adjuntan todo tipo de objetos gráficos cuando sea necesario para ampliar o facilitar las explicaciones.
Además y por norma se utilizará siempre un mismo ejemplo de tres dispositivos con sensores y actuadores.

Se adjuntan también un anexo, al final de la memoria, describiendo como preparar y configurar las herramientas que han sido necesarias en este proyecto.

Capítulo 2

Análisis y diseño del protocolo

2.1 Trama del mensaje

Lo primero y fundamental es hacer un análisis de lo que necesitamos que haga el protocolo para realizar una comunicación entre los dispositivos, de forma que le demos una estructura generalizada al mensaje que se transmite y podamos realizar posteriormente un diseño de los métodos para enviar y recibirlos.

2.1.1 Estructura

Cada mensaje debería contar con campos tales como, identificador del que envía e identificador del que recibe, ya que es un canal de tipo *half-duplex* y todos los dispositivos de la red reciben los mensajes, aunque no estén dirigidos a ellos, por tanto se hace necesario saber que mensajes debemos o no tratar.
Además es necesario también saber quién envía el mensaje que recibimos, pues nosotros debemos responder con otro mensaje (ACK) para confirmar la recepción.

Basándonos en la idea de que hay que responder con un mensaje de confirmación cuando se haga una petición, hay que tener en cuenta que, de recibir dos mensajes del mismo dispositivo, tenemos que confirmar la recepción de cada uno de ellos, por tanto se hace fundamental también un campo que identifique cada mensaje enviado.

Por supuesto necesitamos un campo que defina la acción que queramos que realice el otro dispositivo, que es el objetivo principal por el cual se desarrolla el protocolo.

También y para hacerlo más completo, sería interesante que se pudieran transferir valores en el mensaje, ya sean necesarios para el comando a realizar o simplemente para devolverlos como respuesta a alguna petición, por tanto necesitaremos uno o más campos para estos valores.

Y por último un campo de comprobación que nos asegurará que el mensaje recibido es el inicialmente enviado y que se puede proceder a su procesado.

Aun así y con la estructura ya definida, para darle robustez al mensaje y facilitar su segura transmisión, se pueden añadir campos que indiquen el inicio, el final y otros campos del mensaje, aunque no es imprescindible.
Esto lo vamos a hacer tomando los códigos ASCII que se utilizan normalmente para estructurar los mensajes transmitidos:

	ASCII Symbol	Hexadecimal ASCII Code
SOH	Start of Heading	01
STX	Start of Text	02
ETX	End of Text	03
EOT	End of transmission	04

El primero de ellos será el primer campo de nuestro mensaje, nos indicará que es el comienzo de un mensaje al empezar a recibir la comunicación en el dispositivo.
El segundo de ellos indicará el comienzo de los campos de valores transmitidos en el mensaje.
El tercero de ellos indicará el final de los campos de valores transmitidos.
El cuarto de ellos indicará el final del mensaje.

Con lo que hemos definido antes y el uso de los ASCII code, la estructura del mensaje nos quedaría:

SOH	Indica el comienzo del mensaje
ID Destino	Indica la identificación del destino
ID Origen	Indica la identificación del origen
ID Mensaje	Indica la identificación del mensaje
ID Comando	Indica la identificación del comando a realizar
STX	Indica el comienzo de los campos de valores
… Valores …	Conjunto de valores transferidos
ETX	Indica el final de los campos de valores
Checksum	Comprobación checksum
EOT	Indica el final del mensaje

Aunque en un principio se pudiera pensar que con los códigos STX y ETX sería suficiente para indicar el inicio y final de los campos de valores transmitidos, esto no es así porque ya hemos visto antes que sus representaciones son un 2 y un 3 respectivamente. Si cualquiera de los datos transmitidos fuera un valor numérico 3, estaríamos finalizando la lectura de los valores cuando aún pudiera haber más.

Tomemos este ejemplo:

…	STX	Valores			ETX	…
…	02	45	03	10	03	…

Como vemos, el protocolo tomaría el segundo valor (03) como un ETX y dejaría de leer el tercer valor (10) y se descuadraría el mensaje en el momento de tratarlo ya que luego tomaría el 10 como *checksum* y de ahí en adelante no habría servido de nada definir una estructura.

Esto hace necesario introducir un campo más que defina el número de valores transferidos en el mensaje, así evitamos la situación descrita anteriormente en la que alguno de los valores es realmente un 3.

De forma que finalmente la estructura final sería la siguiente:

SOH	Indica el comienzo del mensaje
ID Destino	Indica la identificación del destino
ID Origen	Indica la identificación del origen
ID Mensaje	Indica la identificación del mensaje
ID Comando	Indica la identificación del comando a realizar
Longitud	Indica el tamaño del conjunto de datos (en bytes)
STX	Indica el comienzo de los campos de valores
… Valores …	Conjunto de valores transferidos
ETX	Indica el final de los campos de valores
Checksum	Comprobación checksum
EOT	Indica el final del mensaje

Con esto resolvemos el problema anterior y pasamos a definir cuantos bytes serán necesarios para nuestro mensaje, en la siguiente sección.

2.1.2 Tamaño del mensaje

- *SOH*: código ASCII que representa el final de los valores transferidos, sólo es necesario 1 byte para representar su valor (01).

- *ID Destino*: es la identificación del dispositivo de destino, aunque con 1 byte pudiéramos identificar hasta 256 dispositivos, para que sea utilizable en instalaciones donde hay un gran número de dispositivos, haremos uso de 2 bytes con el que podremos llegar a representar hasta 65535 dispositivos.

- *ID Origen*: mismo caso que el anterior, usaremos 2 bytes para representar el dispositivo de origen dentro de la red, con hasta 65535 dispositivos.

- *ID Mensaje*: este es un campo con el que distinguir los distintos mensajes que vengan de un dispositivo hacia otro, por tanto y suponiendo que ningún dispositivo vaya a enviar más de 256 mensajes a otro único dispositivo, con 1 byte sería suficiente para representarlo.

- *ID Comando*: para el campo de identificación del comando inicialmente se va a utilizar 1 byte pues estamos hablando de que esto se usaría en instalaciones específicas y no con término general donde hicieran falta miles de comandos adaptables a múltiples configuraciones, por tanto creo que es suficiente el uso de 1 byte y no más para utilizar hasta 256 comandos distintos.

- *Longitud*: este campo define el número de bytes que se usan en el mensaje para transferir los valores, en ningún caso se va a enviar un mensaje con más de 256 bytes, así que usaremos sólo 1 para representarlo.

- *STX*: código ASCII que representa el inicio de los valores transferidos, sólo es necesario 1 byte para representar su valor (02).

- *Valores*: se usarán tantos bytes como se haya definido en la el campo “Longitud”, así que es variable.

- *ETX*: código ASCII que representa el final de los valores transferidos, sólo es necesario 1 byte para representar su valor (03).

- *Checksum*: se usarán 2 bytes para representar el sumatorio de los valores del mensaje para comprobar que el mensaje es correcto. Con 1 solo byte aumenta la probabilidad de que se dé como válido un mensaje erróneo.

- *EOT*: código ASCII que representa el final del mensaje, sólo es necesario 1 byte para representar su valor (04).

Finalmente nos quedaría:

	Número de bytes
SOH	1
ID Destino	2
ID Origen	2
ID Mensaje	1
ID Comando	1
Longitud	1
STX	1
… Valores …	“Longitud”
ETX	1
Checksum	2
EOT	1

En total estaríamos utilizando un mínimo de 13 bytes por mensaje, en el caso de que no se envíen valores.

2.1.3 Posibles valores de cada campo y su significado

Los campos que hemos tratado pueden tomar cualquier valor que pueda representar un byte pero cada valor puede significar algo distinto.

- SOH (1 byte).
  Siempre tiene el valor 1.

- ID Destino (2 bytes).
  - 0
    El mensaje es tipo broadcast, toda la red debe recibir y tratar el mensaje.
  - 1 - 65535
    Identificación del dispositivo al que va dirigido el mensaje.

- ID Origen (2 bytes).
  - 1 - 65535
    Identificación del dispositivo que emitió el mensaje.

- ID Mensaje (1 byte).
  - 1 - 255
    Identificación del mensaje enviado.

- ID Comando (1 byte).
  - 0
    Es un ACK, un mensaje de confirmación.
  - 1 - 255.
    Identificación del comando a realizar.

- Longitud (1 byte).
  - 0
    No hay valores transmitidos en el mensaje.
  - 1 - 255
    Número de bytes de los valores transmitidos en el mensaje.

- STX (1 byte).
  Siempre tiene el valor 2.

- Valores transmitidos (x bytes) (para cada uno de ellos).
  - 0 - 255
    Bytes que definen los valores transmitidos en el mensaje.

- ETX (1 byte).
  Siempre tiene el valor 3.

- Checksum (2 bytes).
  - 0 - 255
    Sumatorio para comprobar la autenticidad del mensaje. *

- EOT (1 byte).
  Siempre tiene el valor 4.

(*) El checksum se hace de los bytes del mensaje excluyendo los bytes de los códigos ASCII (SOH, STX, ETX y EOT), que siempre son iguales y se pueden obviar.

2.2 Proceso de envío

El proceso de envío es posiblemente lo más complicado de todo el protocolo, pues es donde hay que gestionar el acceso al medio y donde tratar de evitar en la medida de lo posible los conflictos.

Hay que tener en cuenta que no es posible habilitar el envío sin anular la recepción, por tanto mientras se esté enviando información no se puede recibir nada.

2.2.1 Mensaje a enviar

Al querer enviar se debe hacer uso de un método al que denominaremos “request”, que se encargará, de forma transparente al usuario, de guardar el mensaje que queremos enviar en una lista de mensajes desde la que será enviado.

Para ello es necesario almacenar los datos de la petición, tales como el destino, el comando a realizar, los valores si los hay… Los datos restantes son calculados o ya se tienen, como puede ser el ID del dispositivo.
Estos datos a enviar deben almacenarse en una estructura a la que llamaremos “message”, que contará con los campos descritos y algunos más que son necesarios:

ID Destino	ID del dispositivo de destino que recibirá el mensaje
ID Comando	ID del comando que realizará el dispositivo del destino
ID Mensaje	Se le atribuye una ID al mensaje
Longitud	Número de bytes necesarios para los valores adjuntos
… Valores …	Valores que se envían adjuntos en el mensaje
Checksum	Checksum
¿ACK?	Indica si necesita un ACK para ser borrado
N. Envíos	Indica el número de veces que se ha enviado
T. Espera	Indica el tiempo (en número de veces) que lleva esperando el ACK

La ID del mensaje es atribuida en el mismo instante en el que se crea la estructura, pues no cambiará hasta que sea eliminado, lo mismo pasa con el checksum, para evitar repetir cálculos en el caso de que haya que reenviar.

Tenemos una variable que indica si es necesario un mensaje ACK para ser borrado de la lista o si por lo contrario se reenviará cuantas veces esté definido.

Si es así, hay una variable que controla el número de veces que se ha enviado el mensaje, ya que habrá un límite dispuesto en la librería por si no responde algún dispositivo, ya sea porque está apagado o no funcional, por lo que se borrará el mensaje después de un número de intentos.

La variable de tiempo de espera es el tiempo que lleva esperando un ACK desde que se envió el mensaje. De alcanzar un límite predispuesto en la librería, pasaría a tratar de enviarse de nuevo, por tanto, es el tiempo que existe entre reenvíos del mismo mensaje.

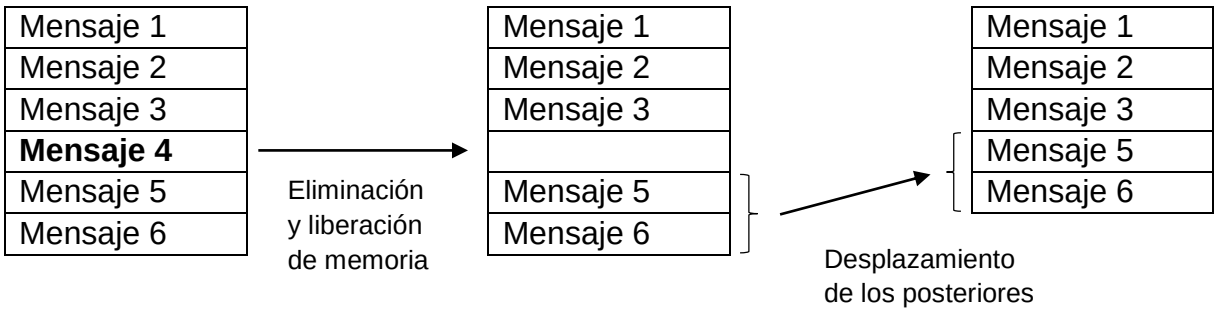
2.2.2 La lista de mensajes

Es necesario contar con una estructura donde guardar los mensajes a enviar pues el envío no se hace en el preciso momento en el que se quiere, primero hay que conseguir obtener el bus y una vez realizado esto, dar salida a todos los mensajes que haya en la lista.

Además es necesario también para mantener vivos los mensajes, puesto que aunque se hayan enviado, hasta que no se reciba la confirmación o ACK si se necesita expresamente, no pueden ser borrados, por si por alguna razón no hubieran llegado al destino y hubiera que enviarlos de nuevo.

La lista puede o no tener un límite de mensajes, aunque lo apropiado y en nuestro caso se realiza una gestión dinámica de ésta para evitar limitaciones. La inserción se realiza siempre por la parte final de la lista. Al eliminarse un mensaje se libera la memoria para que sea reutilizada.

Ejemplo de eliminación de un mensaje (4) en una pila con 6 mensajes de forma dinámica:



2.2.3 Método “transmit”

Se utiliza el mismo método tanto para enviar una petición como para enviar una respuesta ACK, la diferencia reside en el valor del campo “ID Comando” en la trama del mensaje.

El método “transmit” debe ser llamado regularmente para intentar dar salida a los mensajes existentes en la lista.

Lo primero que hará será comprobar si hay algo que enviar, ya que puede no haber ningún mensaje en la lista o todos los que hay se han enviado ya pero permanecen esperando el ACK de respuesta, si es así el método termina.

Lo segundo que hará será comprobar que el bus está libre, para ello se harán los siguientes pasos:

1. Esperar un tiempo fijo, definido en la librería en ms, para dispositivos que tengan una preferencia mayor dada por el número de intentos.
2. Guardar cuantos bytes hay en el buffer de entrada del dispositivo esperando ser leídos.
3. Esperar un tiempo determinado por la identificación del dispositivo y el número de veces de intentar obtener el bus sin éxito (esto otorga prioridad al dispositivo).
4. Comprobar cuantos bytes hay ahora en el buffer de entrada.
5. Enviar tres códigos SOH en intervalos de 1ms para intentar tomar el bus y evitar colisiones, con lectura de los bytes tras cada intervalo.

Hay un primer tiempo fijo, definido en la librería, que están reservados para los dispositivos que lleven ya un número de veces intentando enviar, que tienen preferencia sobre los demás. En este rango, el cálculo del tiempo es en parte aleatorio.

Tras hacer estos pasos hay dos posibilidades, en el caso de que haya más bytes de los que había inicialmente, algún otro dispositivo ha tomado o intentado tomar el bus en este período de tiempo, por lo que salimos del método sin enviar y volveremos a intentarlo más adelante con mayor prioridad.
El otro caso es que sean los mismos bytes y por tanto nadie ha enviado nada en ese período de tiempo, por tanto nos apresuramos a enviar algo por el bus para intentar tomar el control y que los demás dispositivos vean que está siendo utilizado, para ello enviamos hasta tres valores 1 (SOH) y posteriormente, y si no se recibe nada entremedias, empiezan a enviarse los mensajes pendientes.

Esto de esperar un tiempo, y establecer prioridad con ello, sirve para evitar el caso en el que todos los dispositivos sean encendidos a la vez y coincidan al intentar tomar el control del bus. En los demás casos será difícil que coincidan, pero de hacerlo detectarán que hay algún otro dispositivo más enviando y se parará la transmisión, aumentando una variable de la librería que indica cuantas veces se ha intentado tomar el bus, lo que dará aún mayor prioridad para la próxima vez que se intente tomar.

En el siguiente diagrama de actividad se describe el funcionamiento del método a grandes rasgos.

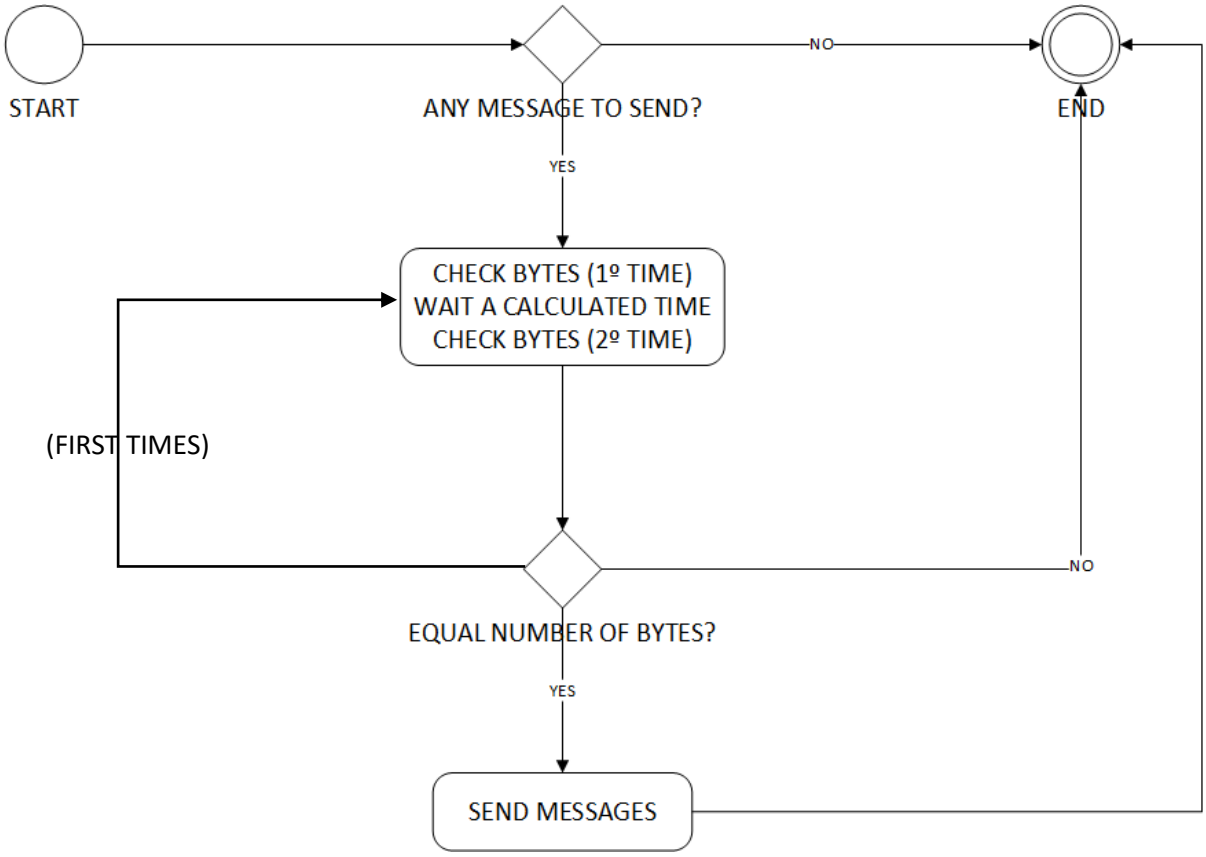


Figura 6. Diagrama de actividad al tratar de enviar

2.2.4 Control del bus

Para controlar el bus es importante especificar un baudrate en la red suficiente para poder enviar información que se propague en menos de 1 ms, para poner el milisegundo como unidad base en la espera mencionada en la sección anterior.
El baudrate normal y más usado es 9600, que equivale a 9600 bits/s o 9600 bits/1000ms o 9.6 bits/ms.

Sería suficiente pues un SOH son 8 bits y se enviaría en menos de 1 ms, pero hay que tener en cuenta también la propagación a través de la red, que ya hemos dicho que puede ser de grandes distancias, por tanto y como la red lo permite sería necesario subir el baudrate para asegurar el envío y la propagación.

Por ejemplo, para un baudrate de 115200 bits/s, que equivale a 115.2 bits/ms, tardaría 0.07 ms en transmitir un byte, con lo que damos tiempo de sobra a la red para que el mensaje se propague, aun así este valor debe establecerse según las características de la red.

2.2.4.1 Peor caso y solución con divisiones

El peor caso es que todos los dispositivos estuvieran sincronizados y pasaran a intentar tomar el bus al mismo tiempo, para ello se realizan tiempos de espera según la identificación y necesidad del dispositivo. Se ha implementado una solución para este caso pero también ayuda a evitar las colisiones en los demás.

Como se dijo anteriormente, esto trataría de realizarse en distintos tramos para que se vayan descartando dispositivos con menor prioridad, lo que permite que en sucesivas ocasiones sólo queden algunos dispositivos intentando tomar el bus y una vez más, su identificación (prioridad) será clave para tomarlo, debe haber 1 ms de diferencia entre distintos dispositivos para que se pueda arbitrar correctamente.

El número de tramos que se van a realizar se extrae del número de divisiones que hacemos según el número de dispositivos que pueda haber en la red. En este caso se va a permitir hasta 65535 dispositivos, pues usamos 2 bytes para la identificación. En la librería hay una constante donde se define el número de divisiones.

Supongamos que tenemos definida la constante con un valor 10, quiere decir que se harán 10 divisiones del número de dispositivos posibles.

Paso 1. 65535 dispositivos disponibles, 10 divisiones, ~6554 dispositivos por división. Cada división espera un tiempo distinto.

- División 1 (entre 1 y 6554) -> Espera 1 ms
- División 2 (6555 y 13108) -> Espera 2 ms
- …

Por ejemplo, si hay dos dispositivos queriendo hacer uso de la red, el nº 4 y el nº 10000, el nº 4 sólo esperará 1ms y enviará una señal para tomar el bus, que tarda menos de 1 ms en propagarse, por tanto cuando termine el nº 10000 de esperar, verá que el bus está en uso y desistirá de intentar tomarlo esta vez.

Paso 2. Seguimos con la primera división entre las anteriores y hacemos otras 10 divisiones.
6553 dispositivos disponibles, 10 divisiones, ~656 dispositivos por división.
Cada división espera un tiempo distinto, igual que antes.

- División 1 (entre 1 y 656) -> Espera 1 ms
- División 2 (657 y 1312) -> Espera 2 ms
- …

Paso 3. Paso 4…

Finalmente conseguiríamos llegar a una división que no contenga más de 10 números, esta será la última y la que tenga más prioridad tomará el bus primero.
En el último caso, supongamos que tenemos tres dispositivos queriendo enviar, con las identificaciones propias 1, 2 y 3 respectivamente, que han llegado a la última fase. El primer dispositivo esperaría 1 ms y trataría de enviar, el segundo dispositivo esperaría 2 ms y trataría de enviar y el tercer dispositivo esperaría 3 ms y trataría de enviar. Como vemos en el siguiente diagrama, el primer dispositivo envía antes que los demás y ellos ya lo saben para cuando terminen de esperar.

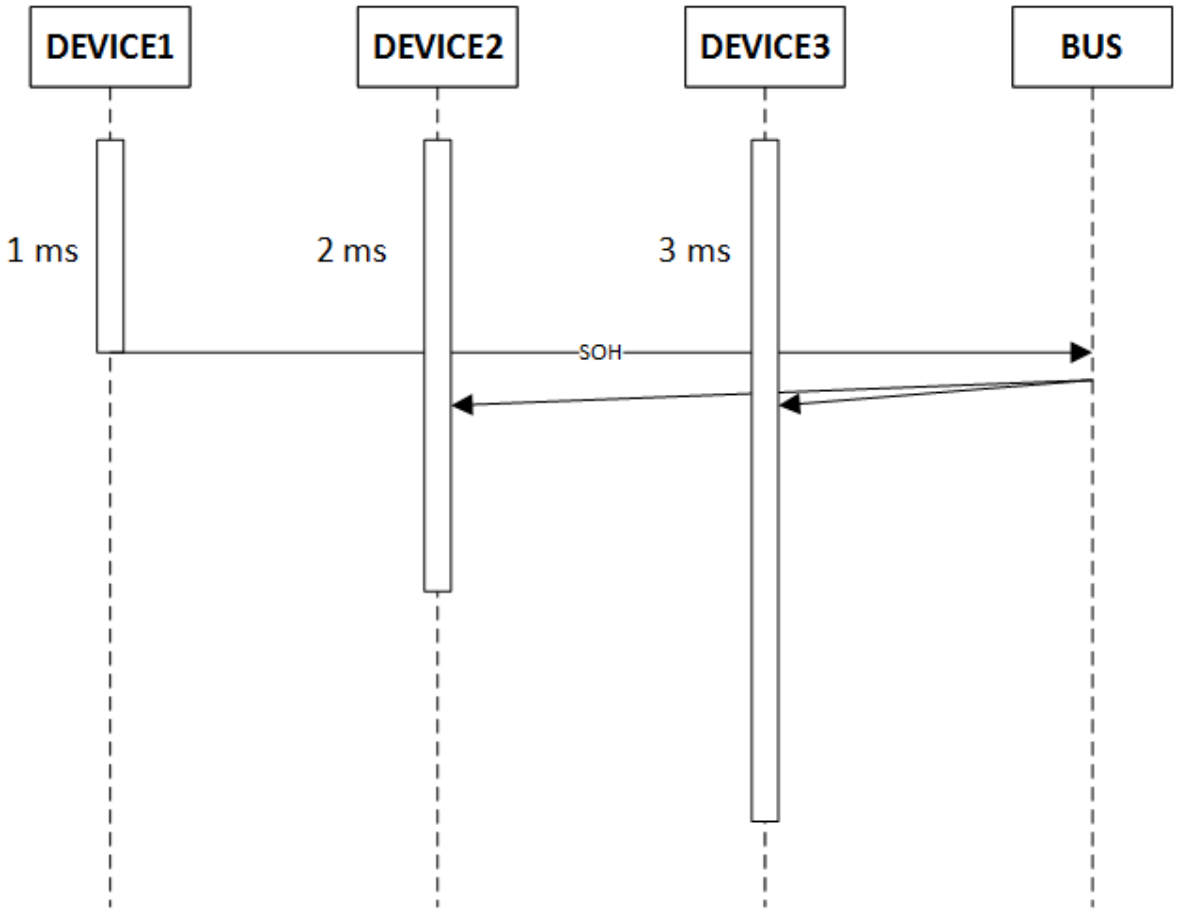


Figura 7. Diagrama de secuencia sobre la diferencia de tiempos de espera

Todos los dispositivos que han visto que el bus estaba tomado cuando han ido a intentarlo ven aumentada su necesidad de tomar el bus, por tanto la próxima vez, aunque pertenezcan a una división inicial simularán ser de una superior división, por ejemplo, si un dispositivo pertenece a la segunda división pero ya ha intentado tomar el bus anteriormente, la próxima vez esperará el mismo tiempo que los de la primera división.

Por encima de todas las divisiones hay un tiempo predestinado a los dispositivos que no consiguen obtener el bus en sucesivos intentos, se le ha llamado división preferente, el tiempo está definido como constante en la librería y se calcula para cada dispositivo de forma aleatoria dentro de ese tiempo.

Estaríamos hablando que, de tener todos los dispositivos sincronizados e intentando tomar el bus al mismo tiempo, el tiempo que tardaría el bus en recibir algún dato sería de entre 0 ms (en caso de que haya algún dispositivo que tenga que enviar preferentemente) y 55 ms (este número sale del número de cifras que tiene el dispositivo con mayor número de identificación más los 5 ms preferentes que nadie va a utilizar en ese caso).

2.3 Proceso de recepción

Cuando un mensaje llega al bus, todos los dispositivos reciben la señal y la introducen en el buffer de entrada del dispositivo, desde este punto se explica cómo se tratan los datos recibidos.

El buffer de entrada tiene un tamaño determinado dependiendo del dispositivo que define cuantos bytes es capaz de almacenar, sería necesario aumentar este tamaño dependiendo del número de dispositivos y por tanto, del tráfico que se espere en la red.

El método que se utiliza para leer el buffer debe ser llamado con más asiduidad que el de transmitir, para liberar bytes en el buffer y que se pueda seguir recibiendo información, lo llamaremos “receive”. Se utilizan fases para saber que parte del mensajes estamos tratando.

2.3.1 Fase 1. Uso de una ventana

En la primera fase se usa una ventana.

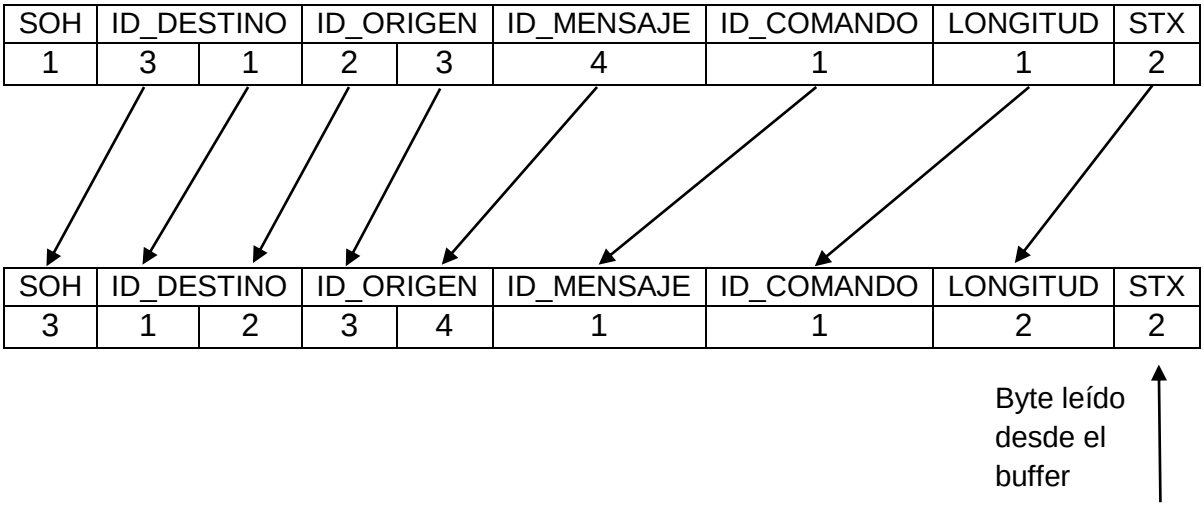
La lectura del buffer de entrada se hace byte a byte, para poder controlar mejor en que parte de la trama nos encontramos para tratarla debidamente, por eso lo primero que vamos a utilizar es una ventana para comprobar que estamos ante un mensaje con una estructura inicial válida.

Recordemos cual es la estructura de un mensaje:

SOH
ID Destino
ID Origen
ID Mensaje
ID Comando
Longitud
STX
… Valores …
ETX
Checksum
EOT

La ventana consiste en tener un array de bytes ya definido de tamaño 9, donde iremos guardando el último byte que hemos leído en la última posición y que nos permitirá comprobar que los 9 primeros bytes son correctos, esto se puede ver en los siguientes ejemplos.

Ejemplo de cómo se inserta en el array (ventana) de 9 bytes.



Como vemos en el ejemplo, todos los bytes se desplazan un lugar en el array y se guarda el byte que leemos en la última posición.
Una vez hecho esto, hay que comprobar que los campos son válidos, tal y como ya definimos en la sección de la trama del mensaje, donde definíamos que valores podían tomar según qué campo.

Ejemplos de arrays (ventana) de 9 bytes incorrectos.

SOH	ID_DESTINO		ID_ORIGEN		ID_MENSAJE	ID_COMANDO	LONGITUD	STX
3	1	2	3	4	1	1	2	2

SOH tiene
que valer 1

SOH	ID_DESTINO		ID_ORIGEN		ID_MENSAJE	ID_COMANDO	LONGITUD	STX
1	1	2	0	0	1	1	2	2

Identificación
de origen no
puede ser cero

SOH	ID_DESTINO		ID_ORIGEN		ID_MENSAJE	ID_COMANDO	LONGITUD	STX
1	1	2	3	4	0	1	2	2

La id del mensaje
no puede ser cero

SOH	ID_DESTINO		ID_ORIGEN		ID_MENSAJE	ID_COMANDO	LONGITUD	STX
1	1	2	3	4	1	1	2	3

STX tiene que
valer 2

Entonces en esta primera fase se comprueban todos estos campos para asegurarnos que al menos en un mensaje válido en sus inicios, suficiente para avanzar a la siguiente fase.

Con esto no podemos decir que el mensaje es válido en su totalidad pero nos evitará hacer cálculos innecesarios en otras fases, y de ser un mensaje inválido en sus restantes partes tenemos un campo “Checksum” para comprobarlo finalmente, así que esto es para evitar trabajo innecesario al sistema.

2.3.2 Fase 2. Guardado de valores adjuntos

En la anterior fase se comprobó que el mensaje es inicialmente válido, en esta fase se tratan los posibles valores adjuntos que se reciben en el mensaje, el número de valores viene definido en el campo “Longitud” que ya tenemos guardado en ese array de bytes (ventana).

Iremos guardando cada uno de los valores transmitidos en un array definido de forma dinámica para su posterior paso al comando que se haya definido en el mensaje.

2.3.3 Fase 3. Final del conjunto de valores adjuntos

Esta fase es la más prescindible de todas pues sólo comprueba que el siguiente byte sea un 3, pues tiene que coincidir con el valor del código ASCII: ETX
Se podría obviar pues ya con la ventana y el posterior “Checksum” se puede asegurar si el mensaje es válido o no.

2.3.4 Fase 4. Comprobación con el checksum

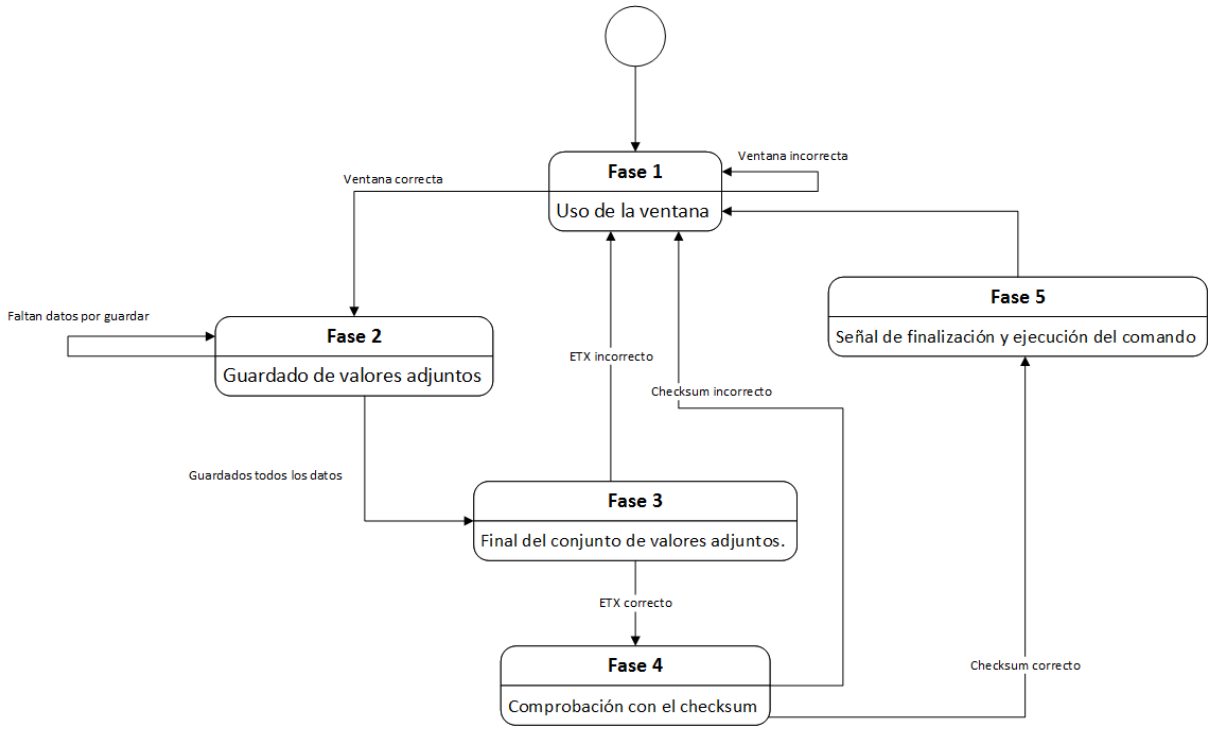
Fase a fase se han ido guardando los valores del mensaje, ahora se realiza el sumatorio de todos ellos en una variable y se compara con el checksum recibido para comprobar que el mensaje es correcto.

2.3.5 Fase 5. Señal de finalización y ejecución del comando

Última fase en la que se comprueba que el siguiente byte es un 4, corresponde con el código ASCII: EOT.
Tras confirmar esto, se dispone a ejecutar el comando pedido pasándole los valores adjuntados si los hay, se termina de efectuar y si se quiere, se devuelve en la misma lista dinámica resultados, así como la longitud de ésta en otro parámetro que tiene el método.

Una vez realizado el comando, se añade a la lista de mensajes un mensaje de tipo ACK para responder al dispositivo que envío el mensaje. El ACK puede contener valores adjuntos o no, dependerá de si el comando devuelve algún tipo de resultado o no, esto debe ser elección del usuario y tiene la posibilidad de hacerlo.

2.3.6 Diagrama de estados de la recepción (fases)



2.4 Comandos y funciones

De nada serviría contar con un protocolo de envío y recepción de mensajes si no queremos hacer nada o no sabemos qué hacer con ellos, por ello es indispensable definir de algún modo qué comandos o funciones tienen que realizarse, tanto al enviar una petición como al recibir un ACK de respuesta.

2.4.1 Registrar un comando

El protocolo tiene que saber que comando es el que quieres ejecutar cuando se recibe un mensaje con una determinada identificación, para ello es necesario registrar el comando deseado para esa identificación.
Para ello se utiliza un método al que hemos llamado “registerCommand”.

El comando será agregado a una lista de comandos, que vendrá definida por programación dinámica.

2.4.2 Dar de baja un comando

En cualquier momento en la ejecución del código del dispositivo, puede ser necesario dar de baja un comando registrado previamente.
Para ello se utiliza un método al que hemos llamado “unregisterCommand”.

El comando será eliminado de la lista de comandos.

2.4.3 Función tras respuesta

Al hacer una petición, dependiendo del método, no sólo se puede recibir un ACK de respuesta, sino que también se puede recibir un mensaje de respuesta con valores. Esto es fundamental para las comunicaciones, pongamos un ejemplo:

Supongamos que quiero que se encienda el aire acondicionado dependiendo de la temperatura que haga en otra habitación, para ello puede ser necesario que el dispositivo que gestiona el aire acondicionado pida la temperatura que hace al dispositivo de la otra habitación y, dependiendo del valor de ésta, encender o no el aire acondicionado. Necesitamos que nos devuelva esa temperatura y ejecutar un método con ese valor.

Por eso al querer enviar un mensaje de este tipo será necesario decir que función ejecutar posteriormente.

Capítulo 3

Implementación y casos

3.1 Clase NET

Se ha desarrollado el protocolo en una clase a la que se le ha denominado “NET” con su cabecera “NET.h” y su correspondiente archivo de código “NET.cpp”.
Se van a explicar todos los métodos de la clase y las variables necesarias.

3.1.1 Constantes en la clase

Las constantes en la clase tienen el prefijo NET delante y no son sólo los ASCII Codes, sino también variables que puede usar el usuario externamente.

Nombre de la constante	Valor
NET_SOH	01
NET_STX	02
NET_ETX	03
NET_EOT	04
NET_BROADCAST	00

Hay otras constantes que definen el funcionamiento del protocolo y que pueden ser adaptadas por el usuario según su conveniencia.

Nombre de la constante	Valor	Descripción
NET_MAX_NTIMES_WITHOUT_ANSWER	10	Número de veces que se esperará la respuesta
NET_NDIVISIONS	10	Número de divisiones a la hora de enviar
NET_FIXEDTIME	10	Tiempo en ms para división preferente
NET_MAX_NATTEMPTS	5	Número máximo de intentos de tomar el bus sin éxito
NET_MAX_NSHIPMENTS	5	Número máximo de veces que se reenviará por la red un mensaje

3.1.2 Tipos de datos y tamaño

Van a ser definidos en puntos posteriores varios tipos de datos que pudieran no estar en todos los sistemas o plataformas, así que en esta tabla se relaciona cada uno de ellos con su tamaño.

Tipo de dato	Tamaño
byte	8 bits / 1 byte
uint8_t	8 bits / 1 byte
uint16_t	16 bits / 2 bytes
boolean	1 bit

3.1.3 Tipos definidos

3.1.3.1 Comandos y funciones

Los comandos y las funciones tienen que cumplir ciertos requisitos.
Un comando que se pueda registrar para el protocolo debe tener dos argumentos:

1. Array de datos transferidos (tipo byte*), se usa tanto para transferir los datos del mensaje al comando como para guardar los valores que se quieren devolver junto con el ACK.
2. Un número (tipo uint8_t) que indique la cantidad de bytes que tiene el array de datos anterior, 0 si no devuelve ningún valor.

Una función que se pueda ejecutar con los valores devueltos de un comando necesita tener los mismos dos argumentos que lo anterior porque, aunque se supone el usuario que configura el sistema preparará una función específicamente para esa respuesta, puede ser útil en caso de devolver una lista de valores.

Los comandos y funciones que sigan estos requisitos serán de tipo void y se les pondrá como nombre de tipo definido: “command”

Al registrar un comando con el método “registerCommand”, se guardará en una estructura definida con nombre “command_t”, que tendrá dos variables.

Nombre de la variable	Tipo de dato	Descripción
_cmd_id	uint8_t	Identificador del comando
_cmd_command	command	Puntero a la función

3.1.3.2 Mensajes

Los mensajes que se guardan en la lista de mensajes tienen que usar una estructura que denominaremos “message_t” y esta estructura tiene que tener unas variables donde guardar lo necesario para enviar el mensaje:

Nombre de la variable	Tipo de dato	Descripción
_msg_idDestination	uint16_t	Identificador del dispositivo al que va dirigido el mensaje
_msg_idMessage	uint8_t	Identificador del mensaje atribuido por la clase
_msg_idCommand	uint8_t	Identificador del comando que se ejecutará en el destino
_msg_length	uint8_t	Cantidad de bytes de los valores transmitidos
****_msg_data***	[byte]	Array donde se almacenan los valores que se enviarán en el mensaje
_msg_checksum	uint16_t	Sumatorio de los campos del mensaje
_msg_ntimes	uint8_t	Número de veces que el mensaje ha podido ser reenviado pero no lo ha hecho porque espera respuesta.
_msg_nshipments	uint8_t	Número de veces que se ha enviado el mensaje
_msg_requireAnswer	boolean	Indica si necesita respuesta o no (*)
_msg_returnCommand	*command	Función a ejecutar cuando se recibe la respuesta

(*) De no requerirla, el mensaje se enviará y será borrado acto seguido. Los mensajes tipo broadcast no se eliminan hasta pasado 5 veces el número máximo de espera de ACKs, pues puede recibir multitud de respuestas.

3.1.4 Variables de la clase

Nombre de la variable	Tipo de dato	Descripción
_serial	Stream	Puerto serie usado para la transmisión
_pin	uint8_t	Pin del dispositivo que controla lectura o escritura
_id	uint16_t	Identificador del dispositivo
_numIdMessages	uint8_t	Número que identificará al siguiente mensaje a enviar
_countCommands	uint8_t	Cuantos comandos hay registrados en la lista de comandos
_countMessages	uint8_t	Cuantos mensajes hay por enviar en la lista de mensajes
_countAttempts	uint8_t	Cuantas veces se ha intentado tomar el bus sin éxito
_rec_phase	uint8_t	Fase en la que se encuentra actualmente el método que trata los mensajes recibidos
_rec_idOrigin	uint16_t	Identificador del dispositivo que envío el mensaje
_rec_idMessage	uint8_t	Identificador del mensaje recibido
_rec_idCommand	uint8_t	Identificador del comando a realizar
_rec_length	uint8_t	Número de bytes ocupados por los valores transferidos en el mensaje
_rec_checksum	uint16_t	Sumatorio de los campos del mensaje para su posterior comprobación
_rec_countData	uint8_t	Contador para saber cuántos valores transferidos llevamos tratados
_window[9]	[byte]	Array donde se almacenan los 9 primeros campos del mensaje
_data	[byte]	Array donde se almacenan los valores adjuntos o transferidos en el mensaje
_sthToTransmit	boolean	Indica si hay algún mensaje para enviar
_messages	message_ptr	Puntero a la lista de mensajes
_commands	command_ptr	Puntero a la lista de comandos

Las variables que tienen el prefijo *_rec* son variables usadas por el método *receive* en particular.

3.1.5 NET()

Al método constructor de la clase se le pasan 3 argumentos:

1 Puerto serie que se usará para transmitir y recibir
2 Pin del dispositivo para controlar la lectura o la escritura en el bus
3 Identificador del dispositivo

En el constructor, se asignan los tres argumentos pasados al método a sus tres variables correspondientes en la clase.

3.1.6 Init()

Hay un método que le hemos llamado “init” para inicializar las variables de la clase por primera vez. No tiene ningún argumento.

3.1.7 Request()

Como se ha explicado en puntos anteriores, y extensamente en el capítulo 2, este método sirve para guardar un mensaje en la lista de mensajes para posteriormente enviarlo. Se le pasan un mínimo de dos argumentos y hasta un máximo de seis argumentos:

1 Identificador del destino, puede ser 0 para tipo de envío broadcast.
2 Comando a ejecutar en el destino
3 (Opcional junto con el 4) Tamaño del array de valores a transmitir
4 (Opcional junto con el 3) Array de valores a transmitir
5 (Opcional) Función a ejecutar cuando se obtenga una respuesta
6 (Opcional) Booleano que indica si esperar o no confirmación del destino

Dado el número de argumentos, tenemos el método principal con los dos primeros argumentos y tres sobrecargas de éste para los argumentos opcionales.

3.1.8 RegisterCommand()

Método que sirve para registrar un comando, para que el protocolo pueda saber que comando ejecutar al recibir un mensaje con esa identificación, tiene dos argumentos:

1 Identificador para el comando
2 Puntero a la función a ejecutar

Si ya hay un comando registrado en la lista lo sobrescribirá.

3.1.9 UnregisterCommand()

Método que sirve para dar de baja un comando registrado, sólo necesita un argumento:

1 Identificador del comando

Si no existe el comando en la lista para el identificador dado, no se hace nada.

3.1.10 Transmit() y receive()

Los métodos para transmitir y recibir mensajes ya han sido explicados en profundidad en el capítulo 2, a ninguno de los dos métodos se le pasan argumentos.

3.1.11 Los métodos en el programa principal de un dispositivo

El protocolo se define en una librería para ser exportado y utilizado en dispositivos, en cualquier plataforma, adaptándola como sea necesario.

Se instancia la clase con el constructor al inicio del programa y se inicializa seguidamente con el método “init()”.
Luego y a ser posible el método “receive()” debería realizarse más veces que el método “transmit()” en el bucle de instrucciones del sistema en el que nos encontremos, para evitar la congestión de la red.
Además el método “transmit()” debería estar aislado de las interrupciones del sistema para evitar quedarse en mitad de un envío.

3.2 Implementación sobre plataforma: Arduino

Para probar el protocolo diseñado hemos hecho uso de la plataforma Arduino, la cual es de código abierto y facilita su implementación. Se han utilizado placas Arduino sobre las que se han instalado placas con sensores y actuadores cedidas por el tutor.

Las placas con sensores y actuadores cuentan con puerto serie rs485 para poder implementar y probar de forma ideal el protocolo desarrollado, estableciendo así una comunicación entre las distintas placas siguiendo el modelo multimaestro.

Arduino se programa en lenguaje C y C++, por tanto la librería del protocolo se ha programado en estos lenguajes. Se ha hecho uso del programa Visual Studio para poder realizar esta implementación, en el anexo 1 se detallan los pasos para poder realizar una implementación de este tipo.

Arduino tiene una rutina de funcionamiento dividida en dos métodos:

- Setup()
  Método que se ejecuta al encender el Arduino y donde deben inicializarse los diversos procesos y librerías que se vayan a usar.

- Loop()
  Método que contiene el código que se ejecutará cíclicamente, es la función que realiza la mayoría de trabajos en Arduino.

Para esta implementación se ha hecho uso de una librería llamada “TimerOne” con la que podemos hacer que se lance una interrupción cada x tiempo y ejecutar un método definido, utiliza el reloj del procesador para ello. En nuestro caso lo hemos utilizado para el método transmit().
La justificación a esta decisión es porque cuando se lanza una interrupción en Arduino, de cualquier tipo, se deja de realizar lo que se esté haciendo en ese momento en el loop(), por lo que si pusiéramos el método transmit() en é, cualquier interrupción pararía la emisión del mensaje por el canal. Además TimerOne es una librería muy utilizada para las programaciones en Arduino, por tanto había que hacer énfasis en ese detalle y explicar cómo debería usarse conjuntamente con la librería.

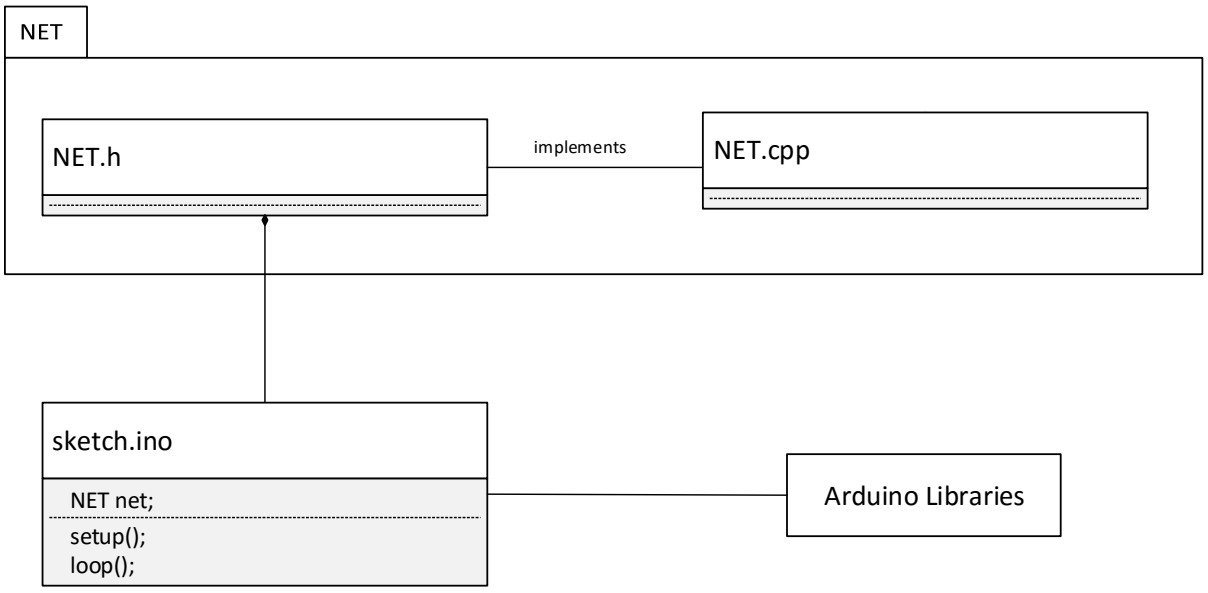
Ya hemos dicho que el método transmit() es mejor relacionarlo con la interrupción del sistema, para que no se pueda parar de ninguna forma la emisión, pero hay que definir donde se ponen los demás:

- La clase NET debe definirse fuera de los métodos de Arduino, al inicio.
- El método init() debe ponerse en el setup() de Arduino
- El método receive() se debe poner en el loop(), pues puede ser interrumpido sin problema y queremos que se realice cuantas más veces posible mejor.

Se ha tomado la decisión de lanzar una interrupción para usar el método transmit() cada 500ms, para reducir el retraso entre la petición de una acción en un dispositivo y la recepción de ésta en otro.

Además se ha cambiado una variable específica en las librerías internas de Arduino para permitir hacer más grande el array del buffer de entrada, el cual por defecto utiliza 64 bytes y pueden resultar pocos, así que se ha aumentado a 256 bytes para permitir un mayor número de mensajes en el buffer.

3.2.1 Diagrama del código utilizado para la implementación en Arduino



3.3 Casos de uso por el dispositivo

Estos son los casos de uso que se pueden dar por parte del dispositivo. Se excluyen las funcionalidades y métodos que funcionan gracias a la librería y el protocolo, sólo se describen los realizables desde el dispositivo en cualquier momento…

3.3.1 Registrar un comando

Descripción	El dispositivo quiere registrar y por tanto identificar una función de éste como un comando del protocolo.
Pre-condición	
Post-condición	La función ahora es un comando dentro de la lista de comandos y está asociada a un identificador único.
Escenario principal	1) El dispositivo ejecuta el método para registrar la función en cuestión. 2) La librería del protocolo introduce el comando en la lista de comandos registrados.
Escenario secundario	
Información adicional	

3.3.2 Dar de baja un comando registrado

Descripción	El dispositivo quiere dar de baja un comando de la lista de comandos del protocolo.
Pre-condición	Debe existir el comando que quieres dar de baja en la lista de comandos.
Post-condición	El comando dado de baja no existe en la lista. La lista de comandos tiene ahora un comando menos.
Escenario principal	1) El dispositivo ejecuta el método para dar de baja el comando dado su identificación. 2) La librería del protocolo encuentra el comando en la lista por su identificación y lo borra.
Escenario secundario	1) El dispositivo ejecuta el método para dar de baja el comando dado su identificación. 2) La librería del protocolo no encuentra el comando en la lista por su identificación y no hace nada.
Información adicional	

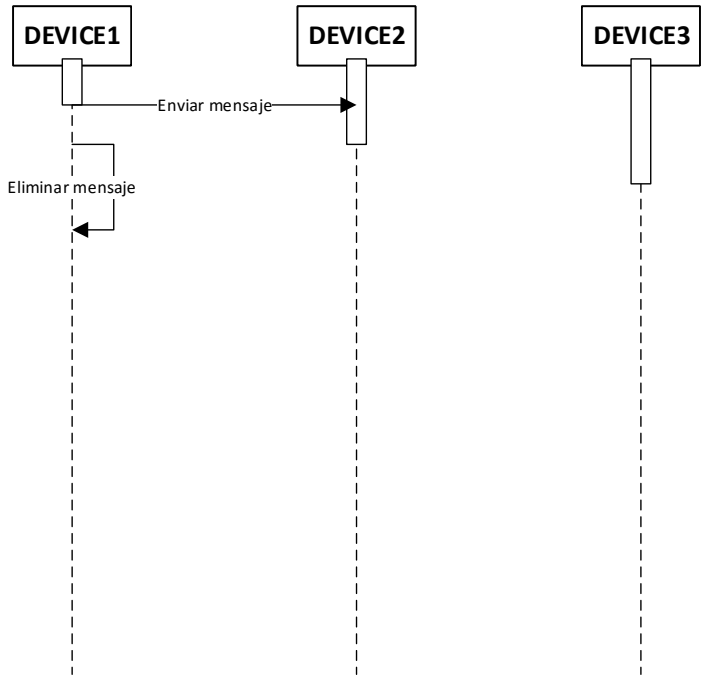
3.3.3 Generar una petición

Descripción	El dispositivo quiere pedir a otro dispositivo que haga algo, así que genera un mensaje o petición.
Pre-condición	
Post-condición	El mensaje está introducido en la lista de mensajes a enviar.
Escenario principal	1) El dispositivo ejecuta el método para generar una petición. 2) La librería del protocolo introduce el mensaje en la lista de mensajes a enviar.
Escenario secundario	
Información adicional	

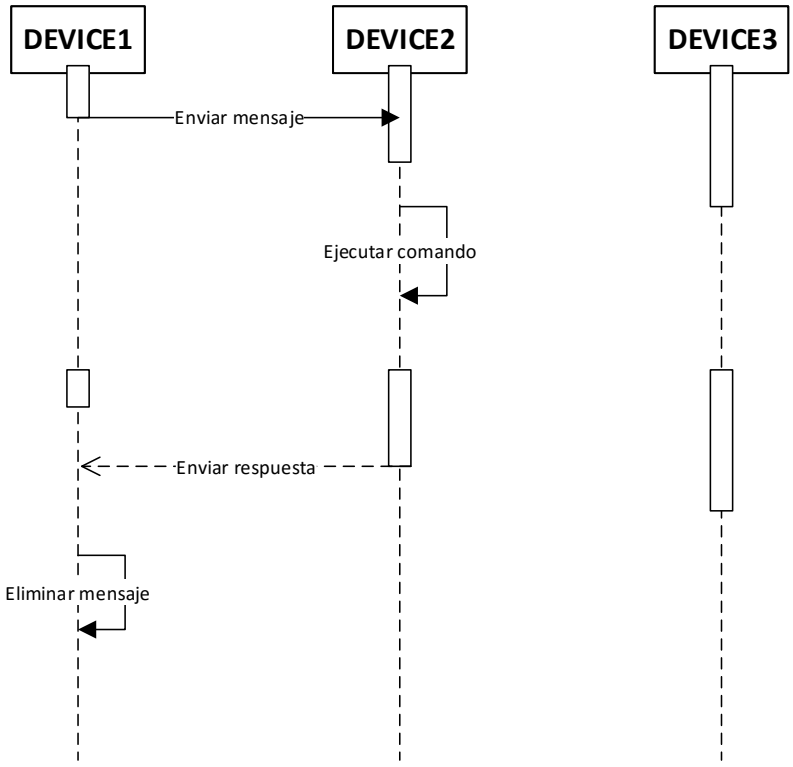
3.4 Casos en el protocolo

Los casos en el protocolo se representarán con diagramas de secuencia, que muestren los distintos escenarios. Para facilitar la comprensión de los diagramas, se utilizará el mismo escenario en todos ellos con tres dispositivos distinguibles:

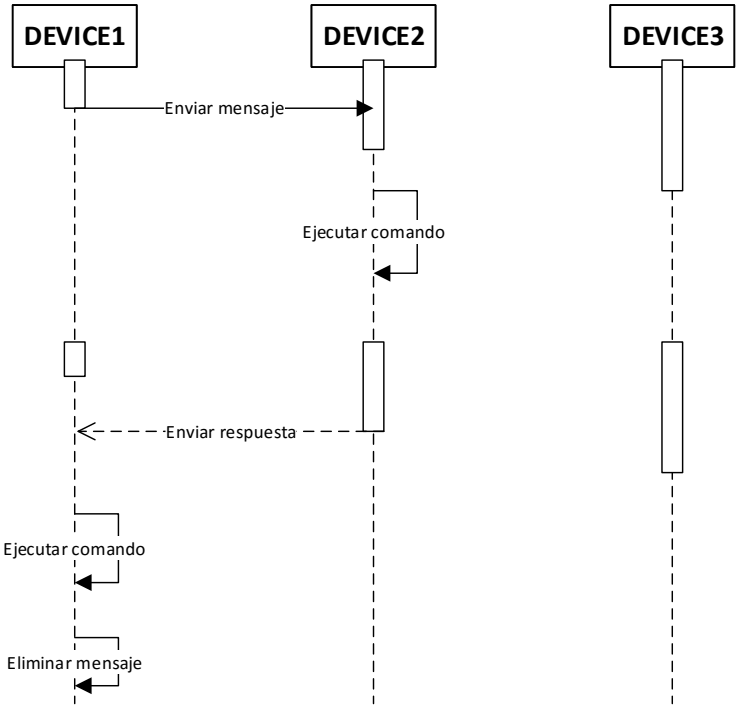
3.4.1 Dispositivo envía una petición a otro dispositivo y no espera respuesta



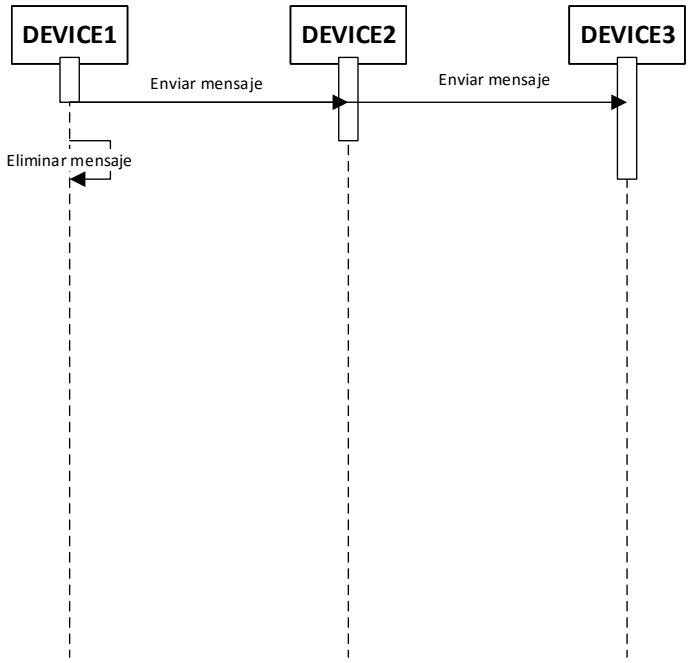
3.4.2 Dispositivo envía una petición a otro y espera respuesta



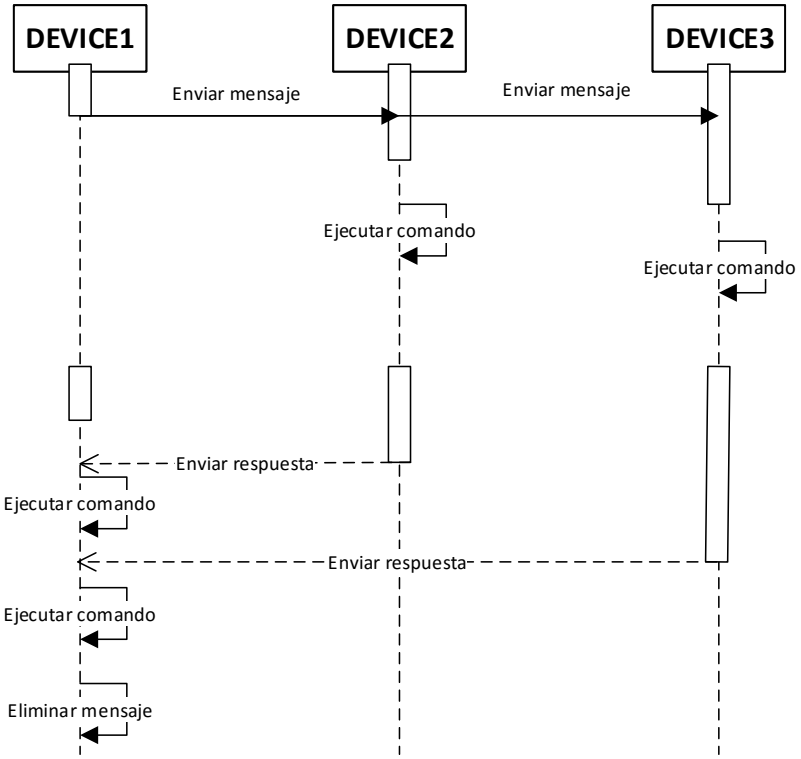
3.4.3 Dispositivo envía una petición a otro y espera respuesta para ejecutar una función



3.4.4 Dispositivo envía una petición broadcast y no espera respuesta



3.4.5 Dispositivo envía una petición broadcast y espera respuestas para ejecutar una función



Capítulo 4

Conclusiones

A lo largo de este proyecto hemos estudiado las posibilidades y limitaciones de los distintos tipos de comunicación según los modos de envío permita el cable que utilicemos. Hemos implementado un protocolo multimaestro para la gestión de las comunicaciones sobre una red RS485 de tipo half-duplex. Esto nos ha permitido adquirir un conocimiento en profundidad de cómo se gestionan las comunicaciones.

El programa realizado es una implementación del protocolo en una librería para sistemas programados en C, pero no nos hemos quedado solo en implementar la librería, sino que ha sido probada con diversos sensores y actuadores en un Arduino como dispositivo. Esta elección para la implementación está justificada por la accesibilidad a este tipo de dispositivo y la facilidad que nos otorga para el desarrollo.

Consideramos este proyecto interesante porque estamos dándole un nuevo enfoque a las comunicaciones que se pueden dar en una red tipo half-duplex como es la RS485, otorgándole versatilidad y dinamismo al permitir que cualquier dispositivo utilice la red de forma directa. Además también abre la puerta a generar nuevas soluciones que puedan valerse de este protocolo.

Este proyecto está englobado dentro de una línea de investigación con la intención de ser utilizado posteriormente para la implementación de las distintas capas restantes del modelo OSI y también para ser utilizado de forma didáctica en asignaturas. Dentro de las posibles líneas futuras de este proyecto, podríamos destacar la implementación de una capa o framework por encima que permita un direccionamiento a IP para cada agente del sistema justo en la capa superior, en el nivel de red.

Bibliografía

[1] José Manuel Huidobro, Ramón Jesús Millán Tejedor: Manual de Domótica. ISBN: 9788492779376

[2] Cristobal Romero Morales, Francisco Javier Vazquez Serrano, Carlos de Castro Lozano: Domótica e Inmótica. Viviendas y Edificios Inteligentes. ISBN: 9788499640174

[3] RS485 specification
www.lammertbies.nl/comm/info/RS-485.html

[4] Información sobre RS-485
https://en.wikipedia.org/wiki/RS-485

[5] RS485: Domótica al alcance de tu mano
http://www.neoteo.com/rs485-domotica-al-alcance-de-tu-mano-15810

[6] Guidelines for Proper Wiring of an RS-485 Network
https://www.maximintegrated.com/en/app-notes/index.mvp/id/763

[7] Explanation of Maxim RS-485 Features
https://www.maximintegrated.com/en/app-notes/index.mvp/id/367

[8] Modelo OSI
https://es.wikipedia.org/wiki/Modelo_OSI

Anexo 1. Manual: herramientas para el desarrollo

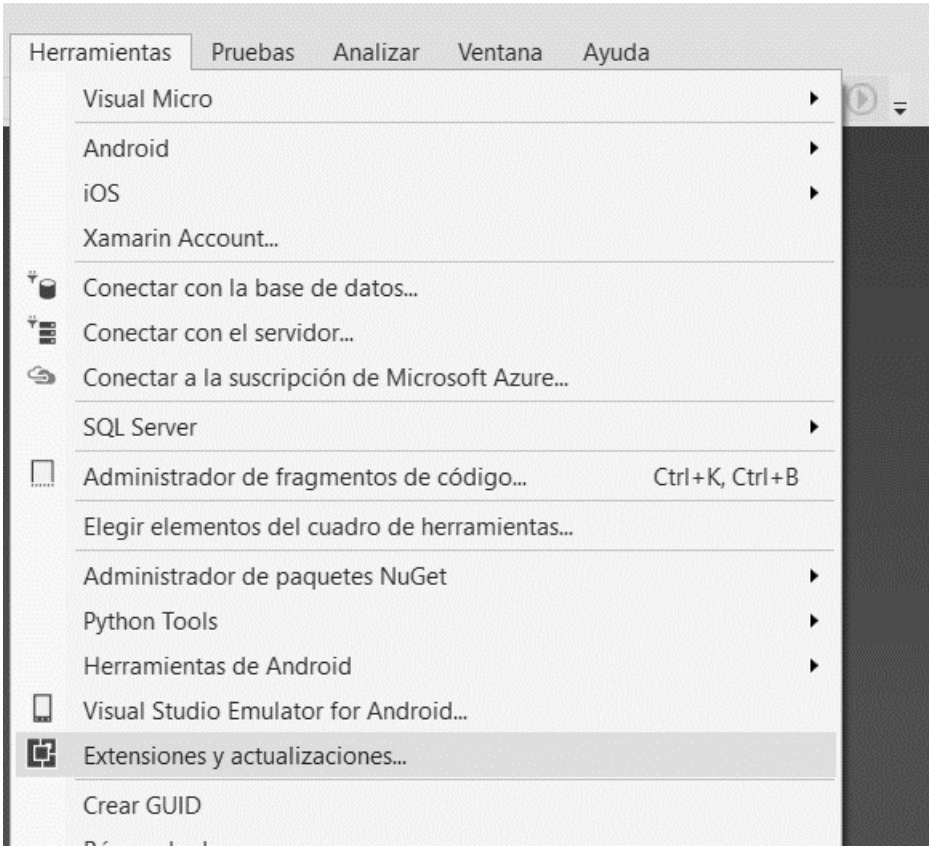
Como herramienta de desarrollo se ha utilizado el programa Visual Studio de Microsoft junto con un complemento (plugin) denominado Visual Micro. Es necesario al menos tener la versión de Visual Studio que incluye C/C++.

Visual Micro (http://www.visualmicro.com) nos da las herramientas necesarias para el desarrollo completo de sketchs de Arduino conectando Visual Studio con las librerías Arduino instaladas en el ordenador.

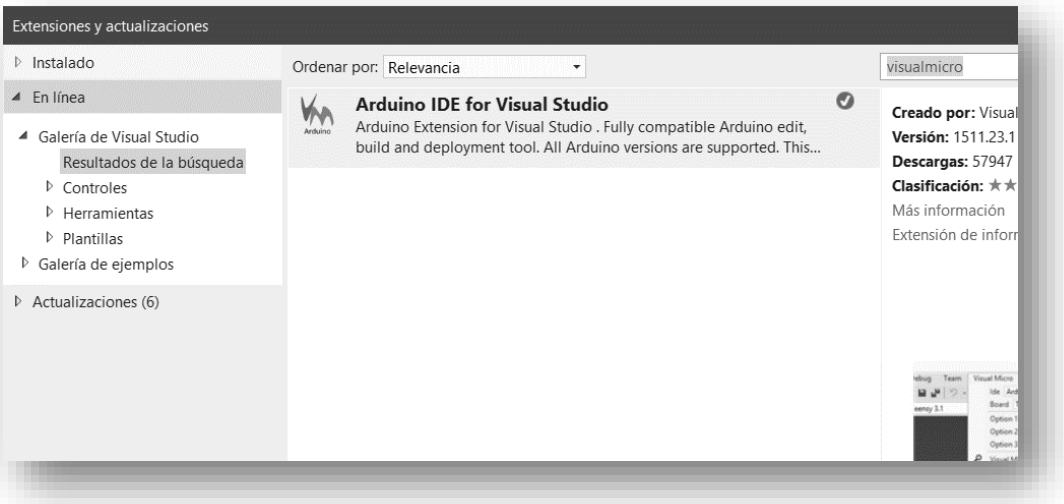
Lo primero es instalar las librerías Arduino desde la página oficial:

https://www.arduino.cc/en/Main/Software

Una vez elegida la versión del sistema operativo que uses y terminadas de instalar las librerías, hay que instalar visual micro en visual studio, para ello haremos uso dentro de Visual Studio del menú “Herramientas” y luego pulsaremos en “Extensiones y Actualizaciones”

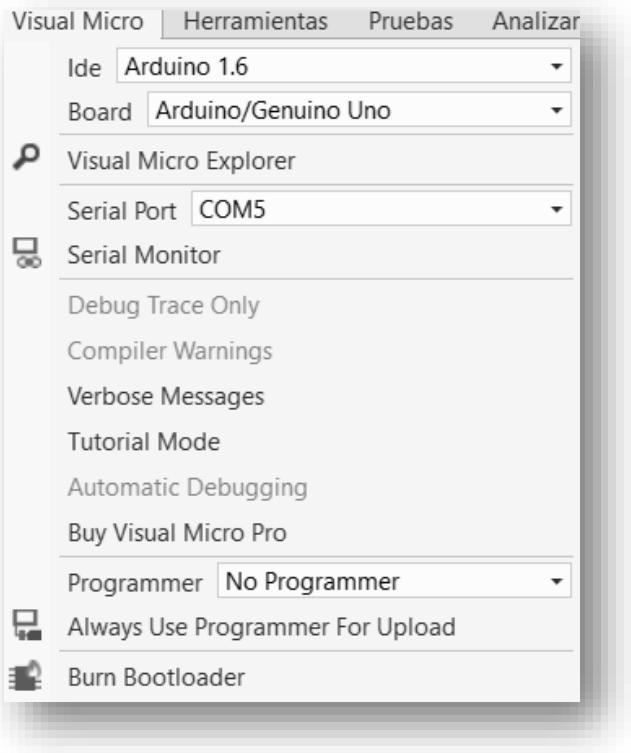


Una vez en la pantalla de Extensiones y actualizaciones, deberemos darle en el lado izquierdo al apartado “En línea” y luego a la derecha arriba, en el buscador, buscaremos: “visualmicro”.

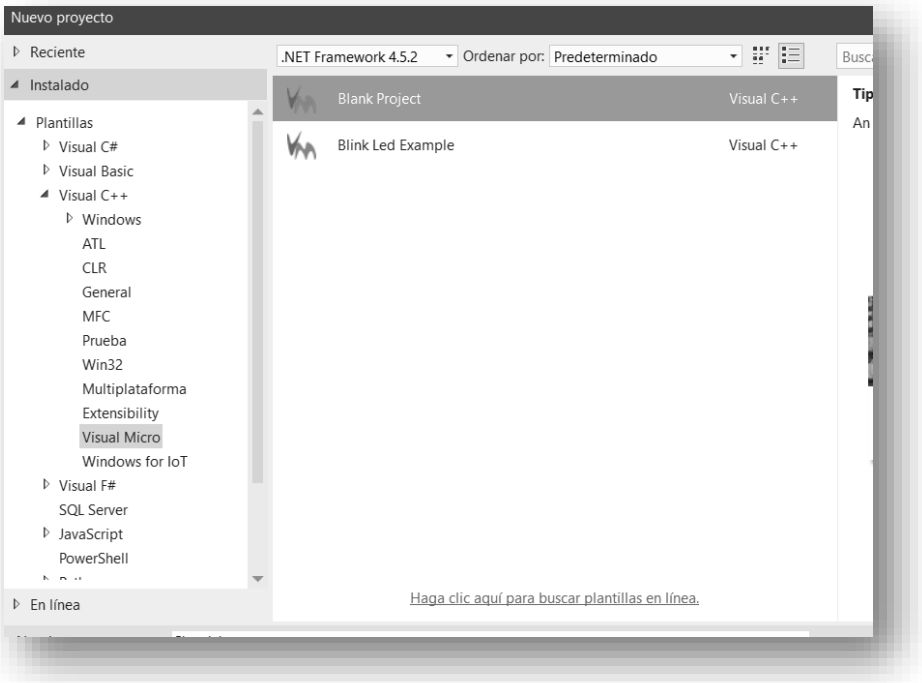


Una vez encontrado le daremos a Instalar.

Con el complemento instalado, nos aparecerá un nuevo menú en nuestra barra de herramientas de Visual Studio.

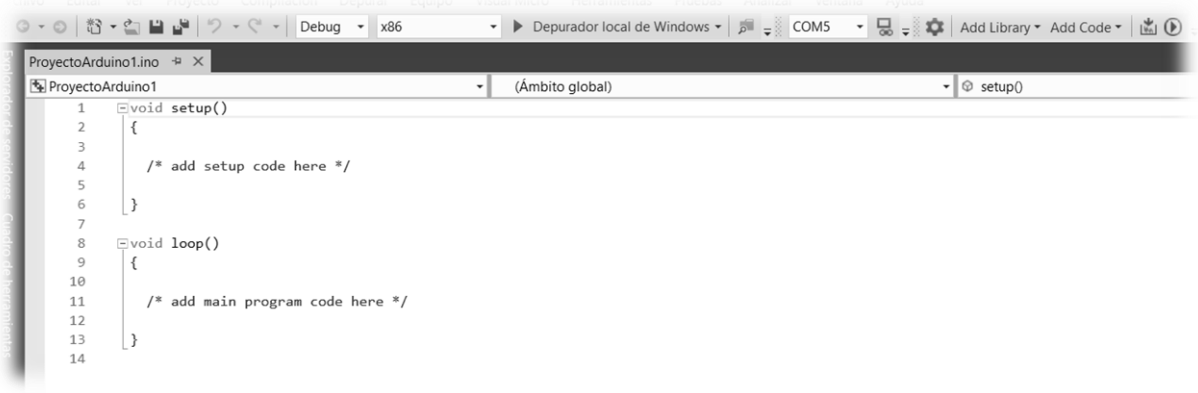


En el podremos definir el ide a utilizar (esto son las distintas versiones de las librerías arduino que tenemos instaladas), la placa que estamos programando, el puerto serie correspondiente con la placa conectada, etc…



Al crear un nuevo proyecto, en el apartado de Visual C++ podremos ver Visual Micro como tipo de proyecto asociado a Android.

Al crearlo obtendremos lo necesario para empezar el desarrollo de nuestro software para Arduino.



Además tendremos disponible una barra de herramientas que nos permite agregar una librería Arduino extra, añadir código desde otro proyecto y los botones para compilar o programar la placa conectada al puerto dado.